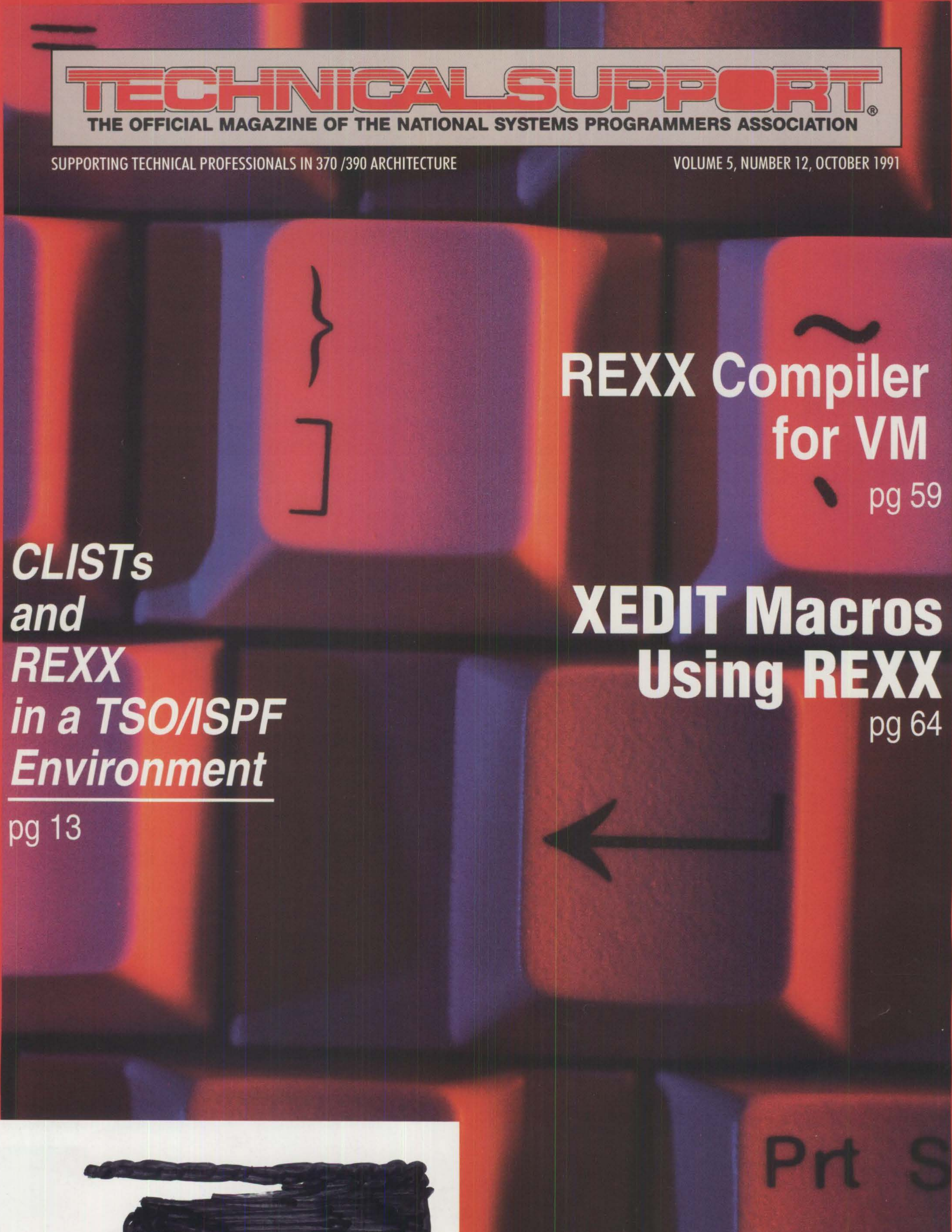
TECHNICAL SUPPORT

THE OFFICIAL MAGAZINE OF THE NATIONAL SYSTEMS PROGRAMMERS ASSOCIATION®

SUPPORTING TECHNICAL PROFESSIONALS IN 370/390 ARCHITECTURE

VOLUME 5, NUMBER 12, OCTOBER 1991



REXX Compiler for VM

pg 59

CLISTs and REXX in a TSO/ISPF Environment

pg 13

XEDIT Macros Using REXX

pg 64

[redacted]

From the President

Things have been hopping around the NaSPA offices these early fall days! I will outline a few of the many upcoming events so that you won't miss any of these opportunities.

The NaSPA Education Foundation (NEF) has a board of directors meeting scheduled for September 28 in Milwaukee. On the agenda for discussion and resolution are November 30 elections and the upcoming educational seminar in San Franciscoto be held December 4, 1991.

The semi-annual Chapter Presidents and Board of Directors Council meeting is on Oct. 11 and 12 in Milwaukee. Each half year we meet to discuss long-term strategic and short-term plans for the association. If you have any input that you would like to have discussed at the council meeting, please contact me or Chapter and Education Coordinator Mary Krukowski.

NaSPA has a board of directors meeting scheduled for October 13, immediately after the council meeting (above). On the agenda for discussion and resolution are any items brought up in the President's Council meeting, the November 30 elections and the 1992 budget and plan.

NaSPA and NEF have their annual elections on November 30. If you would like information on applying for a position on the board of directors of either organization, contact me for an application.

NEF is hosting another technical education seminar! On Wednesday, December 4, 1991, in San Francisco, a one-day seminar on MVS/ESA SP 4.2 Performance Management will be taught by Steve Samson of Candle Corporation. For those of you who have not heard Mr. Samson speak, he is an extremely experienced systems technician and a dynamic speaker. In addition to the course mentioned above, Mr. Samson will also cover IPS/OPT/ICS conversion for SP 4.2, MVS considerations for application software and one of his famous MVS performance free-for-all workshops. The one-day course is $99. You may get additional information from Mary Krukowski.

NASTEC 4.0 is scheduled for Los Angeles next April 1992. If you are interested in being a speaker, the call for papers is currently out. If you are interested in attending, exhibiting or speaking, feel free to contact Debra Lyons, conference manager, at (614) 895-1355.

Chapters: We currently have 27 chapters in the United States and Canada. If you are interested in joining an existing chapter, contact Mary Krukowski for more information. Chapter members receive a 20 percent discount on NaSPA membership dues. Also contact Mary if you are interested in starting a chapter, new chapters are forming in Memphis and Salt Lake City.

That's all for now. Have a great fall!
Sincerely,

Scott P. Sherer
President, National Systems Programmers Association

Technical Support (ISSN 10522581) is published monthly, except for semi-monthly issues in February, August and November by Technical Enterprises Inc., 4811 S. 76th St., Suite 210, Milwaukee, WI 53220-4352. Second-Class Postage paid at Milwaukee, WI. **POSTMASTER:** Send address changes to ***Technical Support***, NaSPA, 4811 S. 76th St., Suite 210, Milwaukee, WI 53220-4352.

Third class mail enclosed

BULK RATE
U.S. POSTAGE
PAID PERMIT 2454
MILWAUKEE, WI

TECHNICAL SUPPORT ®

© 1991 Technical Enterprises, Inc. All Rights Reserved. Published exclusively for the National Systems Programmers Association, Inc.,
4811 S. 76th St., Suite 210 Milwaukee, WI 53220
(414) 423-2420 FAX: (414) 423-2433

VOL. 5, NO. 12, OCTOBER 1991

Vice President, Publishing: John Killoren
Editor: Thomas E. Sprague
Managing Editor: Amy B. Birschbach
Associate Editor/Assistant Production Coordinator: Janice E. Bushey
Production Coordinator /Assistant Editor: Patricia J. Miller
Technical Editors: Paul Bell, Jim McMaster, John Kinne, Mark Hanna, Leo Langevin, Steve Samson, Pete Clark, John Poulakos, Rich ViPond, John Johnston, Paul Waterhouse
Contributing Authors: Richard Brooks, Kelly Brown, Dave Clark, Jeff Furman, Bruce Gaylord, Sam Golob, Billy Guthrie, Mark Hanna, John Johnston, John Kinne, Leo Langevin, Jim McFarland, Gary A. Stotts, Bernard Style
Advertising Manager: Don McMurray Ext. 121
Advertising Sales Representative: Lori Cieslinski Ext. 128
Editorial Assistant: Debbie Flatow

NaSPA DIRECTORY (414) 423-2420

Membership:
Jenny Lucas, Vice President, Membership Ext.106
NaSCOM ID: MBRSHIP
Enrollment Information and Status

•

Administration:
Scott Sherer, President Ext. 104
NaSCOM ID: SHERER
NaSPA Administration

•

Member/Chapter Services:
Mary Krukowski, Chapter/Education Coordinator Ext. 108 NaSCOM ID: CHAPTERS
Chapters, Education Discounts, Education Foundation
Scholarship Applications Ext. 103
Carrie Sherer, Member Services Coordinator Ext. 109
NaSCOM ID: SYSOP
NaSCOM/Telenet IDs, NaSTIP/Tape Orders, Discount Programs
NaSTIP phone orders Ext. 105
Thom Ferris, Sales/Marketing Coordinator Ext. 110
Sponsorship Programs, NaSPA News Newsletter, Job Placement Assistance, Mailing Lists
NaSCOM ID: SALES
Bob Korn, Member Services Manager, Public Domain Tape Development Ext. 117
MasterCard Applications, Ext. 310
NaSCOM ID: SERVICES

•

Convention Services
Debra Lyons, Conference Manager, (614) 895-3440
NASCOM ID: NASTEC

•

Publishing:
John Killoren NaSCOM ID: TECHSUPT Ext. 120
Tom Sprague NaSCOM ID: EDITOR Ext. 122
Amy Birschbach NaSCOM ID: NTHUSED Ext. 123

VTOC/DIRECTORY

On The Cover:

Much of today's data is captured via keyboards. Whether the keyboard is attached to a desk terminal, teller's terminal, airline or car rental reservation terminal or any of the other myriad of keyed data entry devices, this form of data entry for mainframe processing is just as important as the hgh speed optical scanners.

Volume 5, Number 12, October 1991

NaSPA Mission Statement:

The mission of the National Systems Programmers Association, Inc., a not-for-profit organization, shall be to serve as the means to enhance the status and promote the advancement of all professionals using 370 and 390 architecture technical disciplines; nurture member's technical and managerial knowledge and skills; improve member's professional careers through the sharing and dispersing of technical information; promote the profession as a whole; further the understanding of the profession and foster understanding and respect for individuals within it; develop and improve educational standards; and assist in the continuing development of ethical standards for practitioners in the industry.

Multitasking in OS/2:

An Introduction

By John Johnston

Distributed processing has once again surfaced as the "in way" to process and present data. Capitalizing on the strengths of various hardware and software platforms allows corporations to obtain increased power and flexibility for fewer dollars. This increased power and flexibility carries with it a price: increased complexity. As data processing professionals, we must deal with these new complexities and become knowledgeable in various types of computers, operating systems and applications.

A huge debate is brewing in the industry concerning the operating system for PCs in the 90s and beyond. I don't care to debate the issue, but I do feel that with the recent price cut in OS/2 and IBM's renewed marketing push, OS/2 will play a major role in future corporate data processing systems.

When I first decided to learn the internals of MVS, I began by coding assembly language programs that utilized MVS services. I feel that OS/2 will be a strategic operating system, so it is time to delve into OS/2 internals. I do not know of any better way to learn OS/2 internals than to follow the technique I used for MVS.

This article illustrates a PC assembly program that utilizes the services of OS/2 to perform multitasking. The program was written using a 386/SX processor running OS/2 EE Version 1.3. The Microsoft Macro Assembler Version 5.1 was used to assemble the source code. Comparable 370 assembly code is also illustrated for comparison purposes.

The Code

To show how OS/2 allows you to perform multitasking, two PC assembly programs are shown. The programs are PARENT.EXE and CHILD.EXE. The logic and flow of control for these two programs are:

PARENT.EXE

- Start the child program;
- write a message to the screen five times;
- wait for the child to end; and
- exit.

CHILD.EXE

- Write a message to the screen 10 times; and
- exit.

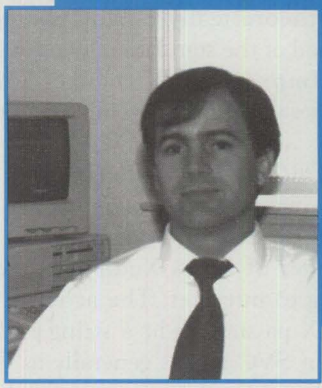
Both of these programs call a subroutine named wait_a_bit to wait a specified amount of time between screen writes.

If you have never looked at a PC assembly program, these source code examples will look a little strange. Don't be intimidated like I was when I first jumped into the language. If you can learn 370 assembly, you can pick up PC assembly.

The Child

Let's look at the OS/2 child program first. See Figure 1. The first four lines give the assembler some information about the program. The program listing is set to 132 characters per line and a title is specified. We do not want to worry about the ordering of the different segments of the program so the assembler is told to make the ordering automatic by the DOSSEG directive. The program will be using instructions that require at least an 80286 processor, so the .286 directive is specified.

The .MODEL directive defines the type of memory model the program will use. The use of memory models and PC memory addressing techniques is fairly complex and a full-blown explanation is beyond the scope of this article. The .MODEL



NaSPA member John Johnston is a mainframe and PC software developer, specializing in peer-to-peer communications, programming and automatic operations.

SMALL statement tells the assembler that one code segment and one data segment will be used by this program.

In the data storage section, 2,048 bytes are reserved for the stack with the .STACK statement. The OS/2 Applications Programming Interface makes extensive use of the stack, so if you are serious about learning PC assembly, you must understand how the PC uses the stack (see the stack sidebar for a discussion on the stack). The start of the DATA segment is defined with the .DATA statement. Beneath it are two data definitions used by the program:

```
message db "Message from Child"
lmessage dw $-message
```

Comparable 370 assembly statements:

```
MESSAGE DC 'MESSAGE FROM CHILD'
LMESSAGE DC AL4(L'MESSAGE)
```

Specified in the beginning of the .CODE segment are the names of three external programs that will be called from this routine. DOSEXIT and VioWrtTTY are OS/2 functions and are comparable to MVS SVCs. The wait_a_bit function will be developed later in this article. These functions are invoked via Call statements and parameters are passed on the stack. DOSEXIT terminates the program, VioWrtTTY writes output to the screen and wait_a_bit puts the program in a wait state for a specified period of time.

Each register in the PC has a specific use. One of these registers, the CX register, is used to hold repeat counts. A 10 is placed into the CX register since we want to execute loop1 10 times. We will see how this counter is decremented later in the program.

To write the message to the screen, the OS/2 function VioWrtTTY is used. This function requires three parameters: the address of the message to be displayed, the length of the message and the output file "handle." A handle is a one-word specification that can stand for a file, the screen or a keyboard. The VioWrtTTY handle for the screen is a zero.

To pass the address of the message requires some knowledge about PC segmented addressing techniques. When an OS/2 .EXE program is started, one of the registers (the DS register) is loaded with a pointer to the DATA segment of the program. To pass an address to VioWrtTTY, the address of the DATA segment and the offset of the parameter relative to the start of the segment must be specified.

Since the full address contains two words (the DATA segment address and the offset) and the stack is a stack of words, two pushes must be issued to pass the address. First the address of the data segment is passed by pushing the DS register. Next the offset of "message" from the start of the data segment is passed with the "push offset" statement. This explanation is oversimplified and the addressing techniques are different between DOS and OS/2. The 80386 and 80486 processors also have their own segmented addressing schemes.

The first four push statements in the child source example show how to pass parameters to a called program. The first two push pass the address of the message, the third passes the length of the message and the fourth passes the screen handle number. All that is left to do is call VioWrtTTY.

Next we want to wait one second. To do this, a one-word number is passed to wait_a_bit containing the number of milliseconds to wait. To wait one second (1000 ms), the constant 1,000 is pushed onto the stack and then the call to wait_a_bit is performed.

To issue the message 10 times, the loop instruction is used. The loop will decrement the CX register, and if CX is not zero, the branch to loop1 occurs. After the program loops through loop1 10 times, the exit code is entered. A zero is pushed onto the

The Stack

The stack is an area in memory that is used extensively by OS/2. There are three main purposes for the stack: to save the calling sequence (by saving the return addresses) of programs and subroutines, to pass parameters between routines, and to use as a temporary work area.

There are two instructions that can be used to place data onto the stack and remove data from the stack. These instructions are PUSH and POP. A convenient way of visualizing the stack is to think of a spring loaded stack of trays in a cafeteria. When you take a tray from the stack, you POP off the top tray. When you return the tray, you PUSH it back onto the stack.

The stack in the PC works in this same manner. Keep in mind that the last tray pushed onto the stack is always the first tray popped off.

When a subroutine is invoked, the CALL instruction pushes the address of the next instruction to be executed in the calling program onto the stack. The subroutine returns to the calling program with the RET instruction. RET uses the address pushed by the CALL instruction to return control to the caller of the subroutine.

To pass parameters between programs, the calling routine pushes either the actual parameter (one word at a time) or the address of the parameter. When the subroutine is called, it will pop these parameters off of the stack. As you can see, it is very important that the caller and the calling routines push and pop the right amount of data from the stack. Failure to do so would cause the RET instruction to use an invalid return address.

/*

stack that tells DosExit to terminate any subtasks that may have been started. Then another zero is pushed that is the completion code of this task and finally DosExit is called. The child program is finished.

• • FIGURE 1: The Child Program

```
;-------------------------------------------------------------------------
;        OS/2 Child Program
;-------------------------------------------------------------------------
         PAGE ,132              ; Set MASM listing to 132 chars
         TITLE OS/2 Child (started by PARENT)
         DOSSEG                 ; Auto segmentation
         .286                   ; Must have at least an 80286
         .MODEL SMALL           ; One data and one code Segment
;-------------------------------------------------------------------------
;        Data Storage Areas
;-------------------------------------------------------------------------
         .STACK 800h            ; 2048 bytes for the Stack
         .DATA                  ; Start of data segment
message  db    "Message from Child"
lmessage dw    $-message        ; Length of message
         .CODE
;-----------------------------
; Tell MASM about OS/2
; routines
;-----------------------------
 EXTRN DOSEXIT:FAR
 EXTRN VioWrtTTY:FAR
 EXTRN wait_a_bit:FAR

start:
;-----------------------------
; Write a message 10 times
;-----------------------------
         mov   cx,10
loop1:
;-----------------------------
; Write the Message
;-----------------------------
         push  ds               ; Pass the DS reg
         push  offset message   ; Pass offset of message
         push  lmessage         ; Pass message length
         push  0                ; Write to screen
         call  VioWrtTTY        ; Write the message

         push  1000             ; 1000 ms equal 1 second
         call  wait_a_bit       ; Call routine to wait
         loop  loop1            ; Loop until CX = 0 (CX is
                                ; decreased each time loop is
                                ; entered)
;-----------------------------------------
;        Exit Code
;-----------------------------------------
         push  0                ; Terminate all of our tasks
         push  0                ; Return code = 0
         call  DOSEXIT          ; And exit

         END   start            ; start = beginning addr (sets CS
```

```
*------------------------------
* MVS Child Program
*------------------------------
CHILD ENTRY
      L     R4,=F'10'
MAIN5 WTO   'MESSAGE FROM CHILD'
      STIMER WAIT,DINTVL=TIMER
      BCT   R4,MAIN5
      EXIT  RC=00
      DS    0D
TIMER DC    C'00000100'
      END
```

WAIT_A_BIT

Wait_a_bit is a simple subroutine that accepts one parameter from the stack. This parameter specifies the number of milliseconds the caller wishes to wait. The first several lines of code in the wait_a_bit routine are about the same as the child routine. The assembler page length is set, a title is specified, and the auto segmentation, the small memory model, the stack area and the code segment are defined.

The procedure label wait_a_bit must be declared PUBLIC. This declaration allows the subroutine to be called by other modules and is accomplished by the PUBLIC directive.

This subroutine will use the OS/2 function DosSleep to place the task into a wait state. This function is declared with the EXTRN statement.

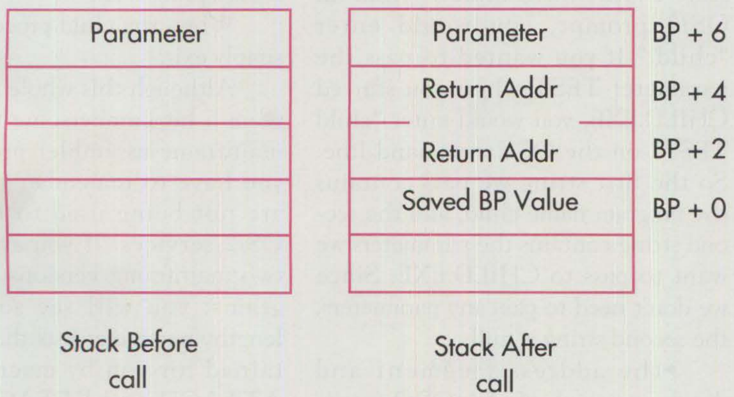
Now on to the heart of the routine. The first thing needed is to set up a base register so we can access the parameter that was passed on the stack. The BP register will be used as the base. The first line of code following the PROC statement pushes the BP register to save its original contents. Next, the stack pointer register (the SP register), which points to the last item on the stack, is copied into the BP register.

Now things get a little tricky. We need to get the parameter that was passed and place it in the AX general purpose register. (Please note that the contents of the AX register will be destroyed since we did not save its original contents.) The following line of code accomplishes this:

```
mov ax,[bp+6]
```

This moves the fourth word from the stack into the AX register. Keep in mind that the stack grows downward into lower memory addresses. At BP+0, there is a one-word (2-byte) value that contains the original value of BP, which was the last thing pushed onto the stack. See Figure 2. When the call to wait_a_bit was

• • FIGURE 2: Passing Parameters on the Stack



made, the call statement pushes the return address onto the stack. (The return address is the address of the next instruction after the call.) Since wait_a_bit is defined as a FAR routine, the return address placed on the stack by the call statement is two words: the segment address and the displacement. So BP+2 and BP+4 contain the return address. Finally, at BP+6 resides the parameter passed from the caller.

Now we are ready to call the OS/2 routine DosSleep to place the task in a wait state. DosSleep requires a two-word parameter be passed via the stack. The first word is the high order wait time, and the second word is the low order wait time. Since our subroutine only accepts one parame-

• • FIGURE 3: The Parent Program

```
;-------------------------------------
;       OS/2 Parent Program
;       - Start the Child Subtask
;       - Issue the Parent messages
;       - Wait for the Child to end
;-------------------------------------
        PAGE ,132  ;Set listing to 132
        DOSSEG     ; Auto segmentation
        .286       ; 80286
        .MODEL SMALL      ; Small model
;-------------------------------------
;       Data storage area (data seg)
;-------------------------------------
        .STACK 800h       ; 2048 bytes
        .DATA             ; Start data
message db  "Message from PARENT!"
lmessage dw $-message  ; Msg Length
buffer  db  256 dup (0) ;Error buffer
command db  "child", 0 ;OS/2 Cmd Line
environ db        ; OS/2 Environment
id_code dw  0         ; Childs PID
exitcode dw 0         ; Childs RC
filename db "child.exe",0 ; filename
term_id dw  0         ; Used by DosCwait

        .CODE         ; Code Seg
;-------------------------------------
;       Tell MASM about the externals
;-------------------------------------
        EXTRN DOSEXIT:FAR
        EXTRN VioWrtTTY:FAR
        EXTRN wait_a_bit:FAR
        EXTRN DosCWait:FAR
        EXTRN DosExecPgm:FAR

start:
;-------------------------------------
;       Start the CHILD task
;-------------------------------------


        push  ds            ; Pass addr of the DATA segment
        push  offset buffer ;    and the offset to the
                            ;    DosExecPgm error buffer
        push  256           ; Pass length of error buffer
        push  1             ; Tell DosExecPgm we want child
                            ;    to run asynchronously
        push  ds            ; Pass addr of the DATA segment
        push  offset command   ;    and offset of the cmd line
        push  ds               ; Pass addr of the DATA segment
        push  offset environ   ;    and the offset of the OS/2
                               ;    environment variables (null)
        push  ds               ; Pass addr of the DATA segment
        push  offset id_code   ;    and a word to hold childs
                               ;    process ID
        push  ds               ; Pass addr of the DATA segment
        push  offset filename  ;    and the offset to filename
        call  DosExecPgm       ; Attach the child task

;------------------------------------------------------------
;       Write the Parent message 5 times and wait 2 seconds
;       between each message
;------------------------------------------------------------
        mov   cx,5             ; Loop count
loop1:
        push  ds               ; Pass addr of the DATA segment
        push  offset message   ;    and the offset of message
        push  lmessage         ; Pass the length of the message
        push  0                ; Tell to write to screen
        call  VioWrtTTY        ; Write the message

        push 2000              ; 2000 ms equal 2 seconds
        call wait_a_bit        ; Call the PROC to wait a while

        loop  loop1            ; Loop until CX = 0 (loop
                               ; decrements CX)

;------------------------------------------------------------
;       Wait for the child to finish processing
;------------------------------------------------------------
        push  1                ; If child started other tasks
                               ;    wait for them also
        push  0                ; Make parent wait until child
                               ;    ends
        push  ds               ; Pass addr of the DATA segment
        push  offset id_code   ; and offset to childs process ID
        push  ds               ; Pass addr of the DATA segment
        push  offset term_id   ; Pass offset to a word where
                               ;    DosCWait will place the
                               ;    terminating task ID (it
                               ;    could be one of CHILDS
                               ;    descendants)
        push  id_code          ; And pass the childs process ID
        call  DosCWait         ; Finally call the wait function
;------------------------------------------------------------
;       Clean up and go home
;------------------------------------------------------------
        push  0                ; Tell OS/2 to terminate all
                               ;    of our tasks
        push  0                ; Return Code = 0
        call  DOSEXIT          ; Call OS/2 to exit
        END   start            ; start = beginning address (sets
                               ;    CS register)
```

```
*-------------------------------------
*        Mainframe Parent Program
*        - Attaches Child
*        - Issues Parent WTO's
*        - Waits for Child subtask
*-------------------------------------
PARENT   ENTRY
         B     START
*-------------------------------------
*        DATA STORAGE AREA
*-------------------------------------
CHILDECB DC    F'0'
CHILDTCB DC    F'0'
         DS    0D
TIMER    DC    C'00000100'
*-------------------------------------
*        ATTACH THE CHILD SUBTASK
*-------------------------------------
START    EQU   *
         ATTACH EP=CHILD,ECB=CHILDECB
         ST    R1,CHILDTCB
*-------------------------------------
*        ISSUE THE PARENT WTO's
*-------------------------------------
         L     R4,=F'5'
MAIN5000 WTO   'MESSAGE FROM PARENT'
         STIMER WAIT,DINTVL=TIMER
         BCT   R4,MAIN5000
*-------------------------------------
*        WAIT FOR THE CHILD
*-------------------------------------
         WAIT  ECB=CHILDECB
*-------------------------------------
*        DETACH THE CHILD SUBTASK
*-------------------------------------
         LA    R1,CHILDTCB
         DETACH (R1)
         EXIT  RC=00
         END
```

ter from the caller (the low order wait time), we first push a zero onto the stack, then push the AX register that contains the value passed to us from the caller. Then we call DosSleep to place the task in a wait state.

When the wait time has expired, DosSleep returns control back to us. Before returning to the caller, the BP register is popped to restore its original value. OS/2 calls adjust the stack to remove the passed parameters so this POP command will pop the right value.

To return to the caller, the RET command is used. Notice that the RET statement has a number following it. This number is used by RET to adjust the stack. The "2" tells RET to remove 2 bytes from the stack before it returns control to the caller. RET also pops the two-word return address so when the caller receives control, the stack is in the same condition as it was before the caller pushed the parameter prior to the call.

The Parent

Now that you know a little about OS/2 functions and passing parameters, the logic of the OS/2 parent program will be simple to follow. The first part of the source code looks a lot like the child. There are a few more items in the data segment and a couple of new OS/2 function programs are defined in the EXTRN section. See Figure 3.

The first thing the parent does is start the child program using the DosExecPgm function. DosExecPgm needs the following parameters:

- the address (segment address and displacement) of a buffer area that DosExecPgm can use to store error messages;
- the length of the error buffer;
- an indicator telling whether the child is to run synchronously or asynchronously. We chose asynchronous for the sample;
- the address (segment and displacement) of two consecutive ASCIIZ strings. (An ASCIIZ string is a string of ASCII characters followed by a zero.) These two strings will make up the OS/2 command line. If you were to start CHILD.EXE directly from an OS/2 prompt, you would enter "child." If you wanted to pass the parameter TEST when you started CHILD.EXE, you would enter "child TEST" on the OS/2 command line. So the first string we pass contains the program name child, and the second string contains the parameters we want to pass to CHILD.EXE. Since we don't need to pass any parameters, the second string is null;
- the address (segment and displacement) of the OS/2 environment area. We will not be using this feature, so we pass the null string located at label "environ" in the source example;
- the address (segment and displacement) of two consecutive words. These words are used when OS/2 terminates the child process. The first word will contain the ID code of child. The second word will contain the return code of the terminating child task; and
- the final parameter we need to pass is the address (segment and displacement) of an ASCIIZ string containing the name of the file we want to execute.

All that's left to do is call DosExecPgm. If all goes well, CHILD.EXE will be started.

The next thing the parent program does is write a message to the screen five times. This code is exactly the same as that in the child program.

After looping through loop1 five times, we want to wait for the child process to complete. To perform this task, we use the DosCwait function. To use DosCwait, a one is pushed onto the stack which tells DosCwait that if the child started other tasks, wait on them also. Next a zero is pushed to tell DosCwait to wait synchronously (wait until the child ends). Next the address of the same double word area used when the child was started is pushed onto the stack. Finally, the address of a word where OS/2 will place the terminating processes ID is pushed along with the child's process ID.

When the child process ends, we simply exit.

Although this whole process may seem a bit cumbersome to seasoned mainframe assembler programmers, you have to remember that macros are not being used to invoke the OS/2 services. If you assemble the two mainframe versions of the programs, you will see some rather lengthy parameter lists that are maintained for you by macros such as ATTACH and DETACH. I don't think it will be long before macros are developed for the various OS/2 services.

To assemble these routines, I set up three files: WAIT.ASM, CHILD.ASM and PARENT.ASM that contain the source code. To invoke the assembler, the following commands were issued:

```
c:\masm wait
c:\masm child
c:\masm parent
```

After all three programs assemble clean, it is time to link them with the OS/2 linker. You must also include the wait_a_bit subroutine and specify the library that contains the OS/2 services by invoking the linker with the following parameters:

```
c:\>c:\os2\link parent wait,,,doscalls
```

and

```
c:\c:\os2\link child wait,,,doscalls
```

Have fun! If you would like to explore deeper into assembly programming in OS/2, I highly recommend a book titled "OS/2 Assembler Language" by Steven Holzner.

/*

Was this article of value to you? If so, please let us know by circling Reader Service No. 86.